AFRL-IF-RS-TR-2005-233
Final Technical Report
June 2005

A BINARY AGENT TECHNOLOGY FOR COMMERCIAL OFF THE SHELF (COTS) SOFTWARE INTEGRITY

Veritas Software Corporation

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J361

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

AFRL-IF-RS-TR-2005-233 has been reviewed and is approved for publication

APPROVED: /s/
ROBERT J. VAETH
Project Engineer

FOR THE DIRECTOR: /s/
WARREN H. DEBANY, JR.
Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 074-0188*

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** June 2005	**3. REPORT TYPE AND DATES COVERED** **Final** **Jul 99 – Jul 04**

4. TITLE AND SUBTITLE A BINARY AGENT TECHNOLOGY FOR COTS SOFTWARE INTEGRITY	**5. FUNDING NUMBERS** C - F30602-99-C-0160 PE - 61101E PR - H563 TA - 10 WU - 01
6. AUTHOR(S) Virin Shah	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Veritas Software Corporation One Hundred Square, Bldg 700 Cambridge MA 02139	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington VA 22203-1714 AFRL/IFGB 525 Brooks Road Rome NY 13441-4505	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2005-233

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Robert J. Vaeth/IFGB/(315) 330-2182 Robert.Vaeth@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT *APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 Words)***

The contractor developed three binary agent core technologies that support the assurance of commercial off the shelf (COTS) software integrity. These technologies are: 1) agent insertion technology, 2) anomaly detection technology, and 3) recovery technology of "First fault diagnosis". This system, called TraceBack, facilitates: 1) anomaly detection and snaps, 2) reconstruction and recovery, 3) automatic diagnosis, 4) distributed tracing with logical threats and timestamping, and 5) Anomaly detection workflow with snap trigger and policies.

14. SUBJECT TERMS COTS, binary, integrity, TraceBack	**15. NUMBER OF PAGES** 15
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

Table of Contents

1 Overview of the COTS Integrity Project

This report is our final technical report for the project. We begin by providing a general overview of project's goals and technical approach. In subsequent sections, we describe the project's technical results, how they address the project's goals, and how they fit in with the project's technical approach.

2 The COTS Integrity Vision

The goal of the COTS Integrity project was to develop technologies for maintaining the integrity of mission critical environments in the presence of COTS software. Our approach inserts software agents directly into COTS binaries. The agents detect anomalous process behavior and data corruption, correct or report problems, quickly recover from these problems, and thereby enhance the overall integrity of COTS software.

By allowing binary agents to be inserted into COTS (commercial, off the shelf) binaries at the deployment site, our project enables systems with high integrity requirements to leverage COTS software for the first time. Most current research focuses on compile time methods which insert various forms of checks into program sources for enhancing the integrity of mission critical environments. Thus, critical environments have often had to forgo COTS software and develop applications that meet their integrity requirements from scratch.

Unlike past approaches based on theorem proving or type checking, our technology for COTS integrity is a *system-level* approach based on binary agents. We developed a system that inserted software agents directly into COTS binaries at the *deployment site* of software applications, whether the sources are available or not. Because the agents reside within the application binary, and are run from within the application environment when the application itself is run, our binary agents monitored the behavior of applications from within, thereby observing, reporting, and sometimes correcting anomalous data or program behavior.

3 The COTS Integrity Approach

Our project focussed squarely on COTS software integrity – in other words, our goal was to maintain the integrity of mission critical environments in the presence of COTS software. Our goal was to discover and to implement to system level approach to detect anomalous process behavior and data corruption, to correct or report problems, to quickly recover from these problems, and thereby to enhance the overall integrity of COTS software.

Our approach for COTS software integrity is based on binary agents. We developed the following three core technologies for COTS integrity.

1. Agent Insertion technology: We developed a technology to insert binary agents into COTS software packages, particularly in the case where the original developers of the software had not anticipated this need. To demonstrate its generality, our platform focus for this research was both PC/NT and mainframe. The agents recorded a trace of execution as the program executed. This trace facilitated recovery.
2. Anomaly detection technology: We developed a technology to enable the inserted agents to automatically detect when the host program is behaving abnormally. Although we implemented default policies, the technology will also allow policy specification by the host at the deployment site. The inserted agents facilitated anomaly detection through mechanisms such as heartbeats and other methods.

3. Recovery technology or "First fault diagnosis": We also developed a technology to allow the application to automatically protect itself from, correct for, recover from, or as a minimum, report the problem by signaling an alarm. To facilitate rapid recovery in mission critical environments, the technology also facilitated rapid isolation of the place in the program where the problem occurred. The traces gathered during live execution and recovered after a fault allowed post-facto debug to isolate a fault without having to rerun the application in a test environment.

   We also had three options for the following:

   (a) extending our work to forensics for distributed systems,
   (b) extending our binary instrumentation system to the SUN/Solaris platform,
   (c) extending our work to automatic diagnosis and remediation.

We will address these extensions in an integrated way in our report. The totality of our system is called Traceback and will be discussed in the following subsections.

3.1 Agent Insertion Technology

Our instrumentation system is based on static binary rewriting. Our project developed binary instrumentation technology for IBM OS390, x86/Windows, SUN/Solaris, and Java. We describe instrumentation using one use of binary agents called TraceBack as an example. This use of binary agents is to record the entire execution history of the program while the program is executing in its production environment, so that if the program suffers a fault the execution history can be quickly used to track where the problem occurred. We note however that we use the same instrumentation technique to insert other types of agents as well.

Binary instrumentation begins by separating code from data. TraceBack relies on known techniques to do so. After separation, code and data are lifted to an abstract graph representation, which is analyzed and modified and then lowered back to a legal binary representation. TraceBack uses published techniques for this process, and uses well-known compiler algorithms like liveness analysis to allow instrumentation code to make use of architectural registers.

3.1.1 Binary Agents or Probes

TraceBack aims to accurately describe the execution history of a program, especially the execution history that directly precedes a crash or program hang. We first describe how TraceBack probes work within a procedure, and then progress to inter-procedural probes. TraceBack supports multiple code modules which are dynamically loaded, so the next section describes cross-module calling.

3.1.2 Intraprocedural Control Flow

We first consider the problem of summarizing control flow in a connected graph of code blocks. In many cases these code blocks will represent a single procedure at source level, but this is not an essential detail. For the moment we limit ourselves to the case where the control flow graph represents a leaf procedure; we consider more complex cases in the next section.

A simple approach to instrumentation is to modify each block to append its address to a trace buffer. While this works, it fails to take advantage of the constrained execution orders imposed by the flow graph. The trace will be accurate, but unnecessarily voluminous at one word per block; this extravagance imposes runtime costs that can be avoided.

We can do better by recording the address of an initial block in a run and then a succession of branch outcomes at block ends. Since the average number of control flow successors for a block is approximately 2, each additional block requires about 1 extra bit of information. This instrumentation scheme splits the probes up into two basic categories: heavyweight probes to start the run, and lightweight probes within a run.

To make this approach workable a few problems must be solved. To keep the lightweight probes simple they cannot involve conditional logic; this means that the number of bits in a trace record available for use by lightweight probes will be limited. The number of blocks that can be described by a trace record is thus likewise bounded. So the heavyweight probes must be placed in such a way that no path through the graph starting at a heavyweight probe can exceed the length limit.

Blocks that end in unconditional branches do not require lightweight probes. Blocks that end in multiway branches will either require special lightweight probes to record the successor block, or, equivalently, one can just end the trace at this point and force all multiway branch targets to hold heavyweight probes.

The limit on run lengths also implies that each loop will contain at least one heavyweight probe (one can do better if the loop trip count is known, but in general it is not known). Also, a heavyweight probe is required at each external entry point to the graph. Because of this, the presence of the heavyweight probes effectively tiles the control flow graph into a set of directed, acyclic subgraphs (DAGs), each headed by a heavyweight probe. Hence, we call the process of identifying the placement for heavyweight probes DAG tiling.

The effectiveness of a DAG-tiling instrumentation scheme depends upon the average path length. Assuming that each trace record is two words long, with the first word being the header block address and the second word reserved for lightweight probe bits, the scheme pays off provided that there are more than two blocks per run, on average.

To support later reconstruction of the control flow, the instrumentation process needs to build up a table of block addresses to DAG Ids, and a table of dag bits to successor blocks. This information is saved out alongside the instrumented executable in a file we call the map file.

Our runtime support library provides a trace buffer to hold successive trace words. We make a pointer to the last written slot in the buffer available. Each heavyweight probe first checks for end of buffer (represented by a special trace record with all bits set), and calls into the runtime to free up space, if at the end of the buffer. The heavyweight probe then writes the DAG Id, pre-shifted, into the trace buffer. Because these probes involve conditional logic, we call them as subroutines. The lightweight probes simply OR their assigned bit as they are executed. For Windows NT binaries on x86, our probes are implemented as shown below. Heavyweight probes are 8 instructions, with two reads (the buffer pointer, and the old next record) and two writes (the updated buffer pointer and the new record).

This probe architecture allows us to collect roughly one line of source code per byte of trace buffer. With a typical buffer of 64Kbytes per thread, TraceBack is able to display tens of thousands of source lines back in time. Furthermore, trace buffers are themselves readily compressible by a factor of 10 or more for ease of archiving or transmission.

3.1.3 Interprocedural Control Flow

If the control flow graph has a call, we can view the return from that call as establishing another entry point into the procedure-level flow graph. Thus, calls are handled simply by placing a heavyweight probe immediately after the call return point.

In practice, this need to break DAGs at calls is a limiting factor for path length. On architectures which make a clear distinction between user and non-user code it is possible to avoid breaking DAGs at calls, provided it is also possible to recover the return addresses from the runtime stack in the event of an exception.

3.1.4 Multiple Threads

Most systems also support multiple, independent threads of control within a process. For tracing it is desirable to keep track of each thread's execution separately, for two reasons: first, it avoids the need to synchronize access to the trace buffer and hence impose artificial execution constraints on the threads; and second, it provides a per thread trace. Most systems with threads also support the notion of thread local storage. We use this to keep a per-thread buffer pointer, and the runtime tries to assign each thread its own trace buffer. Unfortunately, access to thread local storage is typically fairly slow, and usually requires a library call or equivalent. Since we need to access the buffer pointer in our lightweight probe a library call is out of the question.

For Windows NT, we take advantage of the fact that the first 64 TLS indexes can be accessed fairly directly from the TIB, which is pointed to by FS:0x0. Instrumentation assumes that the runtime will be able to reserve TLS index 60 and all probes are set up to directly access this slot, which is at FS:0xF00. If it turns out that this TLS index is not available, the runtime rewrites all the TLS indices in the inline probes using a fixup table, in a fashion similar to the DAG rebasing.

3.1.5 Time and Ordering

The runtime also maintains a notion of ordering. Trace records are inherently ordered by their position within the buffer. To maintain some notion of ordering across buffers, the runtime uses timestamp records.

Timestamps come in two flavors, selectable by the user. The cheap, default flavor is simply a logical clock; the runtime keeps a counter that atomically increments whenever a significant event happens (thread start/end, module load/unload, exception, buffer wrap-around). On platforms that support a high-performance real-time clock, the timestamps can include these values instead.

Our instrumentation heuristically recognizes language artifacts that relate to synchronization or OS services and will automatically insert timestamp probes into binaries at such points. This allows us to reconstruct thread interleavings that are relatively correct for any two trace records A and B in separate threads, we can determine either that A clearly happened before B, B clearly happened before A, or that there was no apparent constraint on the order of A and B.

3.2 Runtime System to Support Recovery

The instrumentation introduced by binary rewriting relies on an external runtime library to provide key services and ultimately to externalize the results of tracing. At a base level, the runtime supports the instrumentation by making a number of trace buffers available for the instrumentation probes to use.

3.2.1 Trace Buffers

Our runtime uses a fairly simple fixed-allocation scheme for managing its resources. At startup, the runtime obtains configuration information which specifies how much memory it should allocate for trace buffers, and how many buffers to create. It then allocates the desired memory and initializes the buffer structures. All the main buffers are the same size. In addition to these main buffers the runtime creates three special buffers. First, the runtime library image contains a small statically allocated buffer that it can assign to threads if dynamic allocation requests fail. Second, the runtime creates probation and desperation buffers. Each buffer is managed as a wrap-around, first-in first-out queue of trace records. At the physical end of each buffer, the runtime writes a sentinel record that triggers a buffer-wrap callback into the runtime from all instrumentation probes.

All threads are initially assigned to the probation buffer, which contains only this sentinel word thus each thread is set to immediately trigger a buffer-wrap the first time it hits instrumented code. This allows us to only allocate the limited set of buffers to threads that are actually in instrumented code.

3.2.2 Buffer Assignment

When a thread buffer wraps, the runtime has an opportunity to assign it to another buffer. The runtime uses a simple first-come allocation scheme: as each probationary thread hits its first probe, it enters the runtime and is assigned to an unused main trace buffer.

If the runtime ever gets to the point where all main trace buffers have been allocated, subsequent threads coming off probation enter a shared desperation buffer, where they periodically continue to hit buffer-wrap and re-enter the runtime (note each thread maintains its own buffer pointer). Since many threads are writing trace records in an unsynchronized fashion into the desperation buffer the trace data itself is not recoverable.

3.2.3 Reusing Buffers

When a thread exits, a thread termination record is added by the runtime and the buffer is freed for subsequent reassignment. The buffer maintains the buffer pointer and the trace records left by the old thread; these will gradually be overwritten by the next assigned thread provided that it is fairly active (though it is not uncommon to see several threads entire lifetimes packed into one buffer).

The runtime may also periodically run a dead-thread scavenging pass to see if any threads have terminated without notifying the runtime (this can happen with abrupt termination of threads). If so, the runtime writes the appropriate post-mortem record and frees the buffer for reassignment.

3.2.4 Sub Buffering

Certain key pieces of state are lost when threads abruptly terminate; in particular, the current buffer pointer is kept in thread-local storage. In general, the runtime cannot reliably locate the most recent trace record in a full buffer.

To handle such cases, the runtime is designed to partition each main trace buffer into a set of sub buffers. Each sub buffer ends with the same sentinels, but the runtime is able to distinguish a full buffer-wrap from a sub-buffer-wrap. At each sub-buffer-wrap the runtime is able to commit the contents of the just-filled sub-buffer (basically, writing the sub buffer's index into the overall trace header), and clear out the next sub buffer so that the thread's progress through the buffer can be easily distinguished by looking for the latest nonzero entry.

Because of the more frequent callbacks to the runtime, and the clearing of the next sub buffer before returning back to the probe, sub-buffering imposes slightly more runtime overhead and discards a bit more of the trace than full buffering. However, sub-buffering allows the system to recover traces from programs that terminate abruptly.

3.3 Anomaly Detection and Snaps

A variety of events trigger the runtime to take a snapshot (snap) of the current trace data and save it to a separate trace file. To properly snap, the snapping thread acquires a lock on the buffers to prevent other threads from making any structural changes, and saves out the trace file to disk.

The snap file includes the raw trace buffers and their trace record contents as well as trace metadata describing details about the process its name, the details about the host OS, the modules loaded into the process, the reason the trace file was generated, and so on. For instrumented modules, the metadata includes a copy of the module keying information, and the actual DAG Id ranges used by the module.

Snaps may also include a memory or object dump, so that TraceBack can display the values of variables or objects at the point of failure.

3.3.1 The Event Thread

The runtime also creates an event thread. This thread remains entirely within the runtime, and has two main purposes, maintaining a heartbeat, and listening for communication from external processes.

The event thread wakes up periodically, attempts to obtains key system locks, and sends a heartbeat status message to an optional external monitoring process. The heartbeat provides us with a simple mechanism to detect hung processes: if the event thread is unable to obtain the locks within a reasonable time period, or is itself blocked, the external watchdog can snap the trace and report a possible hang. The event thread can also look for other predefined triggers, like excessive memory consumption, etc.

The event thread also waits for messages from external processes. These can be status queries or requests to initiate snapping the trace file.

3.3.2 OS/Runtime Interactions

Aside from basic management of trace buffers, there are several difficult and subtle challenges in creating the runtime library.

3.3.3 Gaining Control at Exceptions

The runtime must also gain control at the point of each exception, preferably before the process has had a chance to do any exception handling (first-chance). This gives the runtime the ability to inspect the process state directly at the point of failure, when the forensic evidence is most relevant.

On Windows the runtime intercepts control by rewriting the code that is invoked when an exception is dispatched from the kernel back into the user process. The runtime routine thus has access to the exception context.

In Java, the runtime is not able to gain first-chance control when an exception is thrown. Instead, we wait for the exception to be caught, and process it at that point. To ensure that the exception is caught as early as possible, we inject outer try blocks into every method, handling all exceptions. A special probe is then added to the start of each handler, including the injected ones, to pass the exception along to the runtime. When control returns to an injected hander from the runtime, the exception is then re-thrown so that it continues to propagate as it would have without instrumentation. If there are several instrumented methods on the activation stack, the same exception may be seen by the runtime multiple times; we rely on suppression to ignore the subsequent appearances.

3.3.4 Gaining Control at Termination

To gain control at normal thread and process termination, the runtime hooks the in-process exit points, like ExitProcess in Windows. To distinguish normal from abnormal termination we also hook the so-called last-chance exception handler in Windows, carefully preserving any previously installed handler.

Most systems also allow for abrupt termination of a process, via kill -9 or similar mechanism. These abrupt terminations are often used to terminate hung or deadlocked processes and having a trace of the process at that point can be a valuable aid in understanding why the process was hung. The runtime uses sub-buffering as described above to create recoverable traces. To ensure that these traces are easily externalized, the runtime also will allocate the traces within memory-mapped files. An external agent is thus free to copy the file image at any time.

3.3.5 Using Host Services

The runtime typically requires some access to services of the host OS. At a minimum, the runtime must be able to write files or persist trace data in some other fashion. For maximum flexibility, it is useful to give the runtime fairly broad access to system services. But doing so is tricky, for a number of reasons:

- The calls made by the runtime should not modify the visible state of a process. This typically means not sharing the C runtime library, or carefully saving and restoring shared state like the errno value.
- The runtime can be invoked in unpredictable contexts, especially at exception points. The thread that enters the runtime may hold locks or other critical resources, and a careless call into the OS might cause deadlock. The runtime itself also requires locking.
- The thread that enters the runtime might invoke operations that cause thread synchronizations that did not exist in the original program.
- The thread that enters the runtime might be operating with restricted privileges (say, from client impersonation).
- The runtime might inadvertently invoke instrumented code or cause an exception. Unless precautions are taken, this can lead to an infinite regress.

To deal with these complications in full requires a fair amount of mechanism. A thread must save and restore any shared state, register itself as having entered the runtime (so that any exceptions caused by the thread can be suppressed), attempt to amplify its privilege level, and temporarily switch itself into the desperation buffer so that any trace records generated while in the runtime do not corrupt the user trace.

We typically layer the runtime entries so that they attempt to perform high-frequency, lock-free, risk-free operations without going through this full set of precautions, so that we only pay full price when necessary.

3.4 Reconstruction and Recovery

Trace reconstruction is the process of turning raw trace data into a line-by-line execution trace. Reconstruction requires (1) a trace file, (2) a set of mapfiles from the instrumentation process, one for each instrumented module, (3) (on some platforms) debug information to map from module-relative addresses to source locations, (4) (on some platforms) the instrumented binaries, and (5) source files. The general reconstruction process proceeds in several steps.

3.4.1 Trace Record Recovery

The trace file is examined to verify its integrity. Sub-buffer boundaries are removed to produce a contiguous span of trace data. Each buffer is then mined back-to-front (newest record to oldest) to recover the trace records it contains. These record sequences are then split up by thread, if the buffer housed multiple threads.

Within each thread sequence, each DAG record is checked to determine what module it came from, by extracting the DAG Id and comparing in to the ranges in the trace metadata. When a DAG is resolved to a particular module, the reconstruction algorithm makes a note that the associated mapfile is required to further process the trace (for a variety of reasons, the metadata may describe modules that do not appear in the trace). Consecutive runs of DAGs falling in the same module are grouped into sub-segments. Segments are further broken up by exception and timestamp records.

4 Automatic Diagnosis

While our binary instrumentation system serves as a vehicle for inserting agents, our policy and diagnosis mechanisms must provide methods to find out when things are going wrong. We have made particular progress on a couple of fronts including deadlock detection and memory leaks.

On deadlocks, we track the order in which locks are obtained in data structures kept within our runtime, and detect inconsistent lock ordering, and generate alerts to the operator that they should involve development to fix the problem, all before the first deadlock even occurs. For example, if the program is seen to take out lock A then lock B (then release them) on some occasions, but lock B then lock A on other occasions, we can notify the production operator to contact development immediately.

Within our Java system, we implemented a new form of memory-leak detection instrumentation (more properly known as memory bloat in Java). This instrumentation mode inserts a probe at the allocation site of any object implementing Java's "Container" or "Map" interface, and lets the runtime then track that object's size over its lifetime, using Java "weak references" to track the object without itself becoming a cause for memory bloat. This is integrated with our binary instrumentation system.

5 Distributed Tracing

Distributed tracing stitches together trace data from separate runtimes into a single master trace. At a small scale these runtimes may coexist within a single process, but in general they could come from separate processes or even separate machines.

5.1 Logical Threads

Our distributed trace model looks for RPC-style interactions across entities. Two physical threads that participate in an RPC call-enter-exit-return sequence are fused into a single logical thread for tracing purposes.

We rely on a variety of mechanisms to connect up the discrete parts of logical threads. Each runtime creates a unique ID for itself when it is initialized, using a standard generation technique. If the runtime is able to detect when an RPC call is taking place, it allocates a unique logical thread ID and binds it to the calling physical thread. Associated with this thread ID is a sequence number. The runtime then attempts to augment the RPC payload with a triple of (runtime ID, logical thread ID, sequence number). In some cases, like Java calling native via JNI, this information can be passed directly, out of band. In other cases, like COM, a payload extension can be used to attach the data to the marshaled call arguments. The logical thread Id and sequence number are also written (along with a timestamp) into the current physical threads trace buffer as a SYNC record.

On the receiving end, the callee runtime can access this extra payload. It first looks at the runtime ID, to see if this data comes from a known runtime. If not, the runtime ID is added to the runtime partner list for the callee. Next, the receiving thread is bound to the logical thread, the sequence number is incremented, and a SYNC record is added to its trace buffer. The callee then proceeds on normally. When it hits the RPC return point, the sequence number is incremented yet again, and another SYNC is put into the callee buffer. Return status can also be captured at this time (e.g. a Com HRESULT). The callee runtime ID, logical thread ID, and updated sequence number are sent back as extra payload, and the caller uses these to update its runtime partner table and places one final SYNC into the trace buffer.

The net effect of an RPC call is thus four SYNC records with the same logical thread ID, successive sequence numbers and (assuming synchronized time sources) increasing timestamps, distributed in two separate trace buffers in two runtimes. The runtimes also have exchanged IDs.

If the RPC callee itself makes RPC calls it will likewise pass the logical thread Id along, establishing a causality chain of physical thread trace segments.

5.2 Timestamp Correlation

In cases where it is not possible to augment RPC packets or use out of band signaling, runtimes can fall back on just using timestamps to look for correlated behavior.

Conversely, we can use logical threads to drive timestamp correlation by comparing timestamps in the buffers. This enables us to compute time skew information between machines to align the timestamp clocks.

6 Anomaly Detection Workflow: Triggers, and Policies

To provide a more usable system, TraceBack provides utilities to control and manage the various components of anomaly detection.

6.1 Snap Triggers and Policies

The main items of TraceBack's anomaly detection from a user's perspective are the trace snaps. Users must be able to control when snaps will occur, and how much data they will contain.

TraceBack provides a variety of snap triggers, including program exceptions (language and low-level exceptions, UNIX signals), alerts from associated runtimes such as a memory fault detector, calls to a supplied snap API, and an external snap command to deal with hung or unresponsive processes, hearbeats, etc.

Triggers are controlled by entries in a textual policy file that the runtime reads as it starts up in each instrumented process.

6.1.1 Coordinating Related Processes

In practice, the user's concept of an application may include a group of related processes running on a machine, or across several machines. Sometimes a fault in one of these processes is actually the result of a failure in another of the related processes. TraceBack allows users to configure process groups that are all snapped together if any one experiences a fault. The user can also let TraceBack find groups of related processes.

In order to implement group snaps, each machine hosting TraceBack instrumented processes also runs a separate service process. The TraceBack runtime in each instrumented process communicates with the service using a local protocol, notifying it of snaps, and potentially getting snap requests from the service. Group snaps are not perfectly synchronized, but they're useful in practice, particularly for RPC-style interactions, where the calling thread in one process will suspend as the callee thread in another process executes the remote request.

6.1.2 Snap suppression

It is critical not to produce useless snaps in the first place, since they consume runtime resources to produce, disk space to store, and user attention to analyze and often discard. TraceBack aggressively suppresses snap triggers that appear to be redundant, such as the same exception coming from the same program location. This feature is under runtime policy control, and is a key factor in producing a usable system.

7 Results and Experience with Fault Diagnosis

In practice we have measured our binary instrumentation system TraceBack's overhead on real-life business applications to be much less 5 percent. For example, TraceBack was deployed at PhaseForward Inc. in an environment in which clients used web browsers to interact with a pharmaceutical trials application running on hundreds of centralized servers. TraceBack overhead was measured to be in the low single digits.

The most important use of TraceBack is also the hardest to quantify, namely how does it help software developers understand and fix production problems and bugs. Because a quantitative study would require access to bug fix information that companies do not generally make public, we offer several examples uses of TraceBack for fault diagnoses.

Phase Forward used TraceBack to diagnose an intermittent hang in their production C++ application. The cross-machine traces demonstrated conclusively that the problem was in a third party database connection dll. PhaseForward used this evidence to get a fix from the database company.

A Fidelity Corp. application was not stable in production, and would only stay up for three to four hours at a time. TraceBack revealed that numerous calls to memcpy were overwriting allocated buffers and corrupting neighboring data structures. With the problem narrowed down, the developers were able to fix the worst corruptions giving them days of uptime.

A customer who declined to be identified had an application that crashed after being instrumented. The trace and dump file revealed that the program was passing uninitialized data to a routine, and it gave them the file name and line number of the bug.

At Oracle Corp., TraceBack revealed problems with a Java/C++ application. Application performance was slow because it was taking a large number of Java exceptions. TraceBack revealed that a call to sleep had been wrapped in a try/catch block. The argument to sleep was coming directly from a random number generator, which could return a negative number. When sleep was called with a negative argument, it threw an exception.

Finally, the TraceBack GUI itself was instrumented with TraceBack. While at eBay, one author was looking at traces in the GUI when it became unresponsive. The author took a snap, and sent the trace, in real time, to another author who was back at corporate headquarters. He quickly determined that there was an O(n2) algorithm in the GUI which was making it unresponsive. The engineer at eBay told them on the spot what the bug was and how it would be fixed.

Our experience has shown that we were able to meet these goals. The TraceBack system has been robust. We routinely use it in our own instrumentation system, and ship instrumented versions of our products to customers. Although measured performance overhead on CPU-intensive SPEC applications is about 60 percent, measured overhead on transaction latencies and throughput of real-life business applications is under 5 percent. There is also ample anecdotal evidence that the TraceBack system has been able to facilitate pinpointing production execution bugs, even when the bugs are irreproducible in a test environment.

8 Major Publication

TraceBack: First-Fault Diagnosis by Reconstruction of Distributed Control Flow. (Veritas Corp. Formerly Incert Software Corp.) To appear in PLDI 2005. Andrew Ayers, Richard Schooler, Anant Agarwal, Chris Metcalf, Junghwan Rhee, and Emmett Witchel.